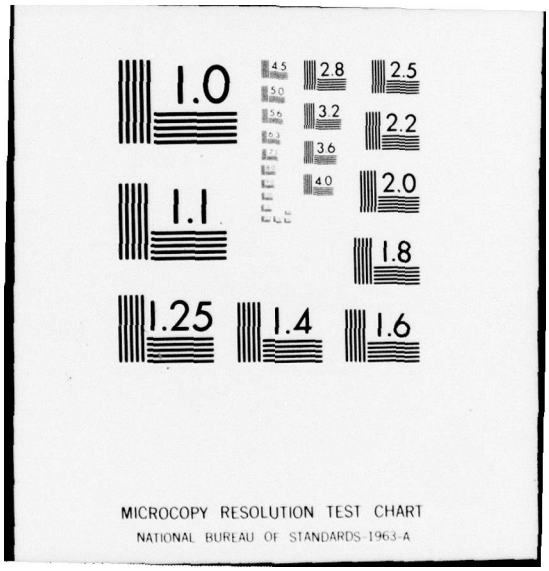
AD-A039 743 FEDERAL COBOL COMPILER TESTING SERVICE WASHINGTON D C F/G 9/2
PROGRAM DEBUGGING USING COBOL 74,(U)
MAY 77 G N BAIRD
UNCLASSIFIED FCCTS/TR-77/14 NL

1 OF 1
AD A039 743

END
DATE FILMED
6-77


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
7.1
8.0
9.0
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

Program debugging using COBOL '74

by GEORGE N. BAIRD
Department of the Navy
Washington, D.C.

INTRODUCTION

The testing and checkout of production software often consumes upwards of 50 percent of the effort that goes into the development of the system.[1] The absence in most programming languages of language elements specifically defined for debugging programs contributes to the time and effort involved in the checkout of systems due to programmers having to use various techniques to improvise debugging code. This was especially true of the 1968 COBOL Standard[2] in that there were no language elements dedicated to the debugging and proving correctness of programs.

The purpose of this paper is to explore the concepts and language constructs introduced into the 1974 COBOL Standard[3] as the Debug Module. What this module has done for the revised COBOL Standard is to provide a means by which the user can describe his debugging algorithm at the source level. This includes the conditions under which data items and procedures are to be monitored during execution of an object program.

For the purposes of this paper, implementor provided techniques will be for the most part ignored. This is due to the wide number of verbs and the various techniques which have emerged as a result of a lack of any formal definition for debugging source statements. The specifications presented in X3.23-1974[3] should provide in a form of standard syntax a semantic composite of implementor debugging techniques. The new debugging module permits the user to determine what to monitor and what information should be provided to the programmer for the actual debugging of the program.

COBOL DEBUGGING WITH X3.23-1968 (AMERICAN NATIONAL STANDARD COBOL)

Prior to the revised COBOL Standard, there were no explicit procedures for specifying what debugging, if any, would take place during the execution of a COBOL program. Although not designated as such, there are language elements in X3.23-1968 that can be used for debugging. The DISPLAY statement, for example, causes low volume data to be transferred to a specific output device. By strategically placing DISPLAY statements in a source program the user could attempt to follow the logic of the program as execution took place. This could be accomplished by DISPLAYing a procedure-name, or a data-name and its contents.

```
PROCEDURE-NAME-1.
    DISPLAY  "PROCEDURE-NAME-1" UPON
    PRINTER.
    NOTE: THE ABOVE STATEMENT INDICATES
    THAT THIS PROCEDURE WAS EXECUTED.
    MOVE 000000 TO COUNT-B
    ADD 1 TO COUNT-B.
    DISPLAY "COUNT-B=" COUNT-B UPON
    PRINTER.
```

The above sequence of statements would show, in the form of output to the printer, that PROCEDURE-NAME-1 was executed each time control was passed to that set of procedures. A few statements later the contents of the data-item COUNT-B would be provided. Execution of the DISPLAY statements would result in the following printed output:

```
⋮
PROCEDURE-NAME-1
COUNT-B=000001
⋮
```

This method of tracing a program's execution is crude at best, and prone to error. For example, once a program has been debugged and is ready to be placed in a production mode, one hardly wants either the debugging information provided or the overhead in the form of the object code generated due to the presence of the debugging statements. Therefore the source code used for debugging the program must be removed. This change in the source program could result in errors being introduced into the program (either syntax or logic) which could necessitate additional testing.

Another consideration in the discarding of debugging statements is the maintenance which might be required during the life of the program. If the requirements for the pro-

gram change, the source program will have to be modified and the need for debugging and testing arises again. The reintroduction of the source statements necessary to accomplish the debugging of the program once again could introduce syntax and/or logic errors. One solution to the problem is to control the execution of the source code for debugging sessions by the use of conditional statements associated with each debugging statement:

```
IF DEBUG-REQUIREMENT EQUAL TO "YES"
  DISPLAY "COUNT-B=" COUNT-B UPON
  PRINTER.
```

The debugging code thus could be executed when needed, and once the program was ready for production the execution of the debugging code could be suppressed. (The control could be in the form of a switch setting, parameter card, value in the Working-Storage.) However, the overhead problems would still exist; the larger program size due to the debugging object code being present, and additional execution time required for testing the debugging requests.

IMPLEMENTOR EXTENSIONS FOR DEBUGGING

The need for debugging functions was satisfied partially by implementors providing language extensions, in their compilers, which were designed for use in debugging the programs, (i.e., TRACE, EXHIBIT, MONITOR statements etc.). The results were inconsistent among implementations as to the name of the verb used and the amount of external control that was available relative to the execution of the debugging statements. These inconsistencies in control included (1) whether the debugging code had to be removed for production runs or the execution of the code could be controlled at execution time, as well as (2) whether the generation of object code could be suppressed if the debugging statements were not removed. The innovative ability of the implementor was certainly the determining factor in this situation as to whether the implementation was an "intelligent" implementation or something with less sophistication.

COBOL DEBUGGING BASED ON X3.23-1974, THE REVISED COBOL STANDARD

The revision to X3.23-1968 has, for the most part, solved the above shortcomings by more realistically providing the tools in COBOL for accomplishing debugging in a reasonable fashion.

The ideal solution, it appears, would be the ability to control, in a way external to the object program, whether the debugging code would be executed. Also, the ability of having the debugging source code remain in the source program and only be compiled when requested would eliminate the problem of source code changes merely for debugging purposes.

A brief description of the capabilities of the debugging module follows:

1. A compile time switch determines whether the debugging source code will be used to generate object code.
2. An object time switch determines, (if the compile time switch caused the debugging statements to be compiled) whether the debugging object code will be executed.
3. Debugging lines are source statements that take on either the characteristics of debugging statements, or comment lines depending on the setting of the compile time switch.
4. The USE FOR DEBUGGING statement specifies the user defined items that are to be monitored by the associated debugging section. The procedural statements in the debugging section define the algorithm by which debugging will take place.
5. A special register called DEBUG-ITEM can be accessed in the debugging sections. This special register contains the source program sequence number of the line in the source program that triggered the execution of the debugging section, the name of the data or procedure or file name involved, and other information relative to the referenced item.

DEBUG MODULE FUNCTIONAL CAPABILITIES

The language elements in the Debug Module of the revised national COBOL standard provide a means by which the user may describe his algorithm to suit his needs. This includes controlling the conditions under which data items or procedures are to be monitored during the execution of the object program. The decision of what to monitor and what information to capture are explicitly in the domain of the user. The introduction of the debug facility merely provides a convenient access to the information pertinent to debugging.

The compile time switch

The capability exists to control, from within the source program, whether debugging source statements are to be compiled or treated simply as comments. The WITH DEBUGGING MODE clause is written as part of the SOURCE-COMPUTER paragraph and serves as the compile time switch over the debugging statements:

```
SOURCE-COMPUTER. Computer-name
  [WITH DEBUGGING MODE].
```

When the optional WITH DEBUGGING MODE clause is specified in a source program, all debugging sections and debugging lines are compiled as regular source statements. The absence of the WITH DEBUGGING MODE clause

causes all debugging lines and debugging sections to be compiled as though they were comment lines.

The compile time switch permits all debugging source code to remain intact in the source program and only be compiled when requested. This satisfies the option of being able to retain all debugging source code in the program for future testing or debugging without having to suffer the overhead that would result from merely suppressing the execution of the object code produced, as discussed earlier.

The object time switch

The object time switch dynamically activates or deactivates the debugging code inserted by the compiler based on the debugging sections described in the declarative portion of the Procedure Division. The accessing of this switch, although dynamic, cannot be accomplished from within the program, i.e., set by a COBOL statement, but is controlled outside the COBOL environment. This will most likely be accomplished through the use of a parameter specified in the operating system control language.

If the switch is 'on' then execution of the program includes the execution of all debugging sections and as a result any output generated by the debugging algorithm specified in the program. The switch can activate the debugging code only if the compile time switch was 'on' (WITH DEBUGGING MODE clause specified in the Source-Computer paragraph) when the program was compiled. Had the compile time switch not been 'on' then no debugging code would have been generated and therefore could not be activated. The ability to control the activation of the debugging code at object time permits the running of the program (if desired) with the debugging code present but dormant. If a problem occurs or spot checking is needed, the program could be run with the object switch 'on' and debugging could continue.

Debugging lines

A debugging line is any line with a 'D' in the indicator area (column 7) of a source line. Depending on the presence or absence of the WITH DEBUGGING MODE clause in the Source-Computer paragraph, the line would be compiled either as part of the source program or as a comment line. (A comment line is defined as any line with an '*' in the indicator area and is ignored by the compiler except for presenting it as part of the source listing.) It should be noted that the object time switch has no effect on debugging lines.

The contents of a debugging line must be such that a syntactically correct program is formed with or without the debugging lines being considered as comment lines. Successive debugging lines are permitted and a statement can be continued on successive debugging lines as long as it results in a syntactically correct source program at compile time. Due to the requirement of the 'D' being present in the indicator area, character-strings may not be broken across two or more debugging lines.

Debugging lines can appear anywhere in a source program after the Source-Computer paragraph. An example of debugging lines follows:

```
      COL 7
        ↓
001400    PROCEDURE-NAME-1.
001500D         DISPLAY "PROCEDURE-NAME-1"
001510D         UPON PRINTER.
001600          MOVE 000000 TO COUNT-B.
001700          ADD 1 TO COUNT-B.
001800D         DISPLAY "COUNT-B=" COUNT-B
001810D         UPON PRINTER.
```

This is the same debugging example illustrated previously for the 1968 COBOL Standard using the DISPLAY statement, but here control is achieved over whether the coding will be considered for compilation (compile time switch).

Since debug lines can appear in both the Environment and Data Division, as well as the Procedure Division, not only procedural statements can be associated with debugging, but data and files as well. For example if the output from the debugging session were to be sent to a temporary file and examined later for some reason, the file itself could be described in such a manner that it would only be compiled into the program when the compile time switch was on.

```
002000    ENVIRONMENT DIVISION.
                ⋮
003000    INPUT-OUTPUT SECTION.
003100D         SELECT DEBUG-FILE ASSIGN TO
                TAPE.
003200          SELECT PAYROLL-FILE ASSIGN
                TO DISK.
                ⋮
004000    DATA DIVISION
004100    FILE SECTION
004200D   FD DEBUG-FILE LABEL RECORDS
004210D   OMITTED.
004300D   01 DEBUG-REC PIC X(200).
004400    FD PAYROLL-FILE
004500        LABEL RECORDS STANDARD.
                ⋮
005000    PROCEDURE DIVISION
005100    INITIALIZATION SECTION
005200    HOUSE-KEEPING
005300D         OPEN OUTPUT DEBUG-FILE.
005400          OPEN INPUT PAYROLL-FILE.
                ⋮
006000D         WRITE DEBUG-REC.
                ⋮
007000D         CLOSE DEBUG-FILE.
```

In the above example a file which is to contain debugging information, and all of the references to it are defined as D lines, and their presence or absence is controlled through the compile time switch.

The USE FOR DEBUGGING statement

For tracing procedure-names and the activity of data items, the debug line would be woefully inadequate or at best pedestrian in nature. The extensive tracing of procedures and contents of data items is accomplished through the use of debugging sections.

The USE FOR DEBUGGING statement is a declarative statement and defines a debugging section. This permits the programmer to establish a debugging section and to identify items that are to be monitored by the associated debugging section. The rest of the declarative procedures in the section define the algorithm by which debugging will take place:

```
DECLARATIVES
  DEB-1 SECTION. USE FOR DEBUGGING ON
                 ALL PROCEDURES.
  PARAG-1.
                 ⋮
  DEB-2 SECTION. USE FOR DEBUGGING ON
                 DATA-NAME-1.
                 ⋮
  END DECLARATIVES.
```

A debugging section is executed each time, as appropriate, that the named item(s) (whether named implicitly or explicitly) are referenced elsewhere in the Procedure Division. A debugging section allows the user to provide a set of procedures for acting on the data provided in DEBUG-ITEM. The contents of the DEBUG-ITEM are updated by the debug system each time a reference is made to an item for which debugging has been requested.

The COBOL debugging statements themselves produce no debugging output. The debugging section permits the user to analyze what is taking place and selectively take any necessary action which may include producing printed output. This is opposed to the 'shotgun' approach of producing all data provided by the debugging system. It also provides an advantage over the TRACE, MONITOR, EXHIBIT, . . . statements in that not only is more information available, but it is provided to the user in the debugging section rather than being sent to a printer-destined file. The user has the ultimate decision as to what, if anything, is to be printed. The TRACE statement could be simulated very simply:

```
DEB-1 SECTION. USE FOR DEBUGGING ALL
               PROCEDURES.
DEB-PARA-1.
    DISPLAY DEBUG-ITEM.
```

The reference to the special register 'DEBUG-ITEM' is the method of accessing the data that is provided to the user by the debug system. The contents of this data item will be covered fully later, it is sufficient to note that one of the fields contains the name of the procedure that was just referenced/entered/altered.

There are basically four options or types of references that can act as triggers in a debugging section. The following discussion shows the various forms of the USE FOR DEBUGGING statement that can be used and the effect of using each.

The first option of the USE FOR DEBUGGING statement allows for the tracing of the execution of the various sets of procedures in the procedure division (paragraph trace):

```
USE FOR DEBUGGING ON {ALL PROCEDURES       }
                     {procedure-name-1 . . .}
```

There are two methods of specifying the procedures on which to trap, as is shown in the above USE statement. When the procedure-name-1 phrase is chosen, the debugging section is executed immediately before each execution of the named procedure and immediately after the execution of any ALTER statement which references procedure-name-1. In this case, each procedure-name on which debugging will take place must be specified in a USE statement. (A USE statement may contain a reference to more than one procedure-name.) On the other hand, if the ALL PROCEDURES phrase is specified, then the above described action takes place for each procedure-name in the program; the only exception would be procedure-names specified in debugging sections.

The second option of the USE FOR DEBUGGING STATEMENT is for monitoring the activity of data items referenced in the Procedure Division. This includes monitoring a data item either when its contents have been modified or each time it is referenced. (For the purposes of this discussion the term "identifier" represents a data-name in a COBOL program including all qualifiers necessary to make it unique and all subscripts/indexes required to reference it.)

```
USE FOR DEBUGGING ON
  [ALL REFERENCES OF] identifier-1.
```

If the optional ALL REFERENCES phrase is specified, the debugging section is executed for every statement that explicitly or implicitly references identifier-1. The absence of the ALL REFERENCES phrase causes the debugging section to be executed immediately after the execution of any COBOL statement that references identifier-1 and causes the contents of the data item referenced by identifier-1 to be changed.

There are a few isolated cases in which an implicit reference to an identifier can cause a debugging section to be executed. A case in point is the WRITE statement where the FROM phrase is used:

```
WRITE record-name FROM identifier
```

results in implicit reference to identifier in that it is moved to record-name prior to the record actually being written as though a MOVE statement had been used. This results in an implicit reference to an identifier causing a debugging section to be executed.

The third form of the USE FOR DEBUGGING state-

ment is for referencing file-names:

USE FOR DEBUGGING ON file-name-1 . . .

This debugging section would be executed after any explicit reference to the named file, (e.g., OPEN, CLOSE, READ, DELETE, START). There is a mild inconsistency here due to the syntax and structure of the COBOL language. This statement will not cause the monitoring of all of the activity of the file due to the WRITE and REWRITE statements. With the exception of the WRITE and REWRITE statements, all other statements that are used in file processing explicitly reference the file-name (i.e., READ file-name, OPEN I-O file-name, etc.). The WRITE and REWRITE statements reference the record-name for the file which is an identifier. Therefore, in order to trace all of the activity of a file, each of the records associated with that file must also be referenced in a USE for debugging. This could be done by using one or more USE statements.

There is a fourth option of the USE FOR DEBUGGING statement which is for use in debugging programs utilizing the COBOL teleprocessing capability; it will not be discussed in this paper.

DEBUG-ITEM

The previous paragraphs described the method of establishing a debugging section for procedure-names, identifiers, and file-names. When a debugging section is executed (i.e., the item being monitored is appropriately affected) the debugging system must provide the information necessary for the programmer to debug the program. This is done through a special register generated by the compiler when the compile time debugging switch is 'on'. This special register is accessible in the debugging section by the name of DEBUG-ITEM.

There are six fields subordinate to DEBUG-ITEM which provide the debugging information:

1. DEBUG-LINE—Contains the line number of the source line which caused the debugging section to be executed.
2. DEBUG-NAME—Contains the first thirty characters of the name that caused the debugging section to be executed, e.g., procedure-name-1, identifier-1, or file-name-1. This would include qualifiers and/or subscripts if present but truncated at thirty characters.
3. DEBUG-SUB-1, DEBUG-SUB-2, DEBUG-SUB-3—There are three occurrences of this item which, and in the case of a data-name which was subscripted/indexed, will contain the occurrence number of each level of the referenced table.
4. DEBUG-CONTENTS—Contains information relative to the name that caused the debugging section to be executed.

- Identifiers—contains the contents of the identifier.
- File-name—contains spaces except that after a READ statement it contains the contents of the record read.
- Procedure-names—contains a variety of information indicating for the most part how control was passed to the procedure-name contained in DEBUG-ITEM:
  START PROGRAM
    (first paragraph).
  SORT INPUT
  SORT OUTPUT
  MERGE OUTPUT
  PERFORM LOOP
  FALL THROUGH
  GO TO
  USE PROCEDURE
    (other than debugging).

The COBOL description of DEBUG-ITEM could be presented as follows:

```
01 DEBUG-ITEM.
   02 DEBUG-LINE PIC X(6).
   02 FILLER PIC X VALUE SPACE.
   02 DEBUG-NAME PIC X(30).
   02 FILLER PIC X VALUE SPACE.
   02 DEBUG-SUB-1 PIC S9(4) SIGN IS LEADING
        SEPARATE CHARACTER.
   02 FILLER PIC X VALUE SPACE.
   02 DEBUG-SUB-2 PIC S9(4) SIGN IS LEADING
        SEPARATE CHARACTER.
   02 FILLER PIC X VALUE SPACE.
   02 DEBUG-SUB-3 PIC S9(4) SIGN IS LEADING
        SEPARATE CHARACTER.
   02 FILLER PIC X VALUE SPACE.
   02 DEBUG-CONTENTS PIC X(n).
```

The X(n) described in the Picture for DEBUG-CONTENTS is the same length as the identifier being debugged or in the case of the procedure-name the length necessary to hold the content of the information placed there by the debugging system.

The following example demonstrates the use of debugging sections for procedure-names and identifiers as well as the results that could be expected.

EXAMPLE:

```
089000 PROCEDURE DIVISION
090000 DECLARATIVES.
090100 SECT1 SECTION.
090200      USE FOR DEBUGGING ON
              PARAGRAPH-X.
090300 DEB-1.
090400      DISPLAY DEBUG-ITEM.
090500 SECT2 SECTION.
090600      USE FOR DEBUGGING ON COUNT-B.
090700 DEB-2.
090800      DISPLAY DEBUG-ITEM.
090900 END DECLARATIVES.
091000 FIRSTSECT SECTION.
```

```
091100 PARAGRAPH-1.
091200     OPEN OUTPUT PRINT-FILE.
       ⋮
101100     MOVE 000000 TO COUNT-B.
101200 PARAGRAPH-X.
101300     ADD 000001 TO COUNT-B.
       ⋮
102100     IF COUNT-B LESS THAN 2 GO TO
             PARAGRAPH-X.
102200     CLOSE PRINT-FILE.
102300     STOP RUN.
```

The results of the above execution of coding would be:

DEBUG LINE	DEBUG-NAME	DEBUG-CONTENTS
101100	COUNT-B	000000
101200	PARAGRAPH-X	FALL THROUGH
101300	COUNT-B	000001
102100	PARAGRAPH-X	GO TO
101300	COUNT-B	000002

In the previous example, had the USE FOR DEBUGGING ON PARAGRAPH-X statement contained the ALL PROCEDURES phrase and the USE FOR DEBUGGING ON COUNT-B statement contained the ALL REFERENCES phrase then the appropriate debugging section would have been executed each time COUNT-B was referenced and for each procedure-name referenced in the program. The following would have been produced:

DEBUG LINE	DEBUG-NAME	DEBUG-CONTENTS
091100	PARAGRAPH-1	START PROGRAM
101100	COUNT-B	000000
101200	PARAGRAPH-X	FALL THROUGH
101300	COUNT-B	000001
102100	COUNT-B	000001
102100	PARAGRAPH-X	GO TO
101300	COUNT-B	000002
102100	COUNT-B	000002

CONCLUSION

We conclude this discussion by offering several thoughts with regard to improved software reliability and increased programmer productivity.

1. The advent of new language elements which are designed for debugging in COBOL will permit consideration for the debugging of source programs to be included in the design and programming of an application—not as an afterthought. For knowledge of potential problem areas would permit the inclusion of both debugging sections and debug lines which could provide ample information for producing an adequately "debugged" program.
2. The use of source program preprocessors and assorted post processors including "core dumps" should be for the most part eliminated from the list of tools necessary for the debugging of COBOL programs now that a high level of control can be accomplished through the use of COBOL source statement elements.

REFERENCES

1. Boehm, Barry W., *Some Information Processing Implications of Air Force Space Missions: 1970-1980*, The Rand Corporation, RM-6213-PR, January 1970.
2. *X3.23-1968 American National Standard COBOL*, American National Standards Institute, Inc., 10 East 40th Street, New York, New York, 10016.
3. *X3.23-1974 American National Standard Programming Language COBOL*, American National Standards Institute, Inc., 10 East 40th Street, New York, New York, 10016.

BIBLIOGRAPHIC DATA SHEET	1. Report No. FCCTS/TR-77/14	2.	3. Recipient's Accession No.
4. Title and Subtitle Program Debugging Using COBOL 74			5. Report Date 9 May 1977 6.
7. Author(s) George N. Baird			8. Performing Organization Rept. No.
9. Performing Organization Name and Address Federal COBOL Compiler Testing Service ADPE Selection Office Department of the Navy Washington, D. C. 20376			10. Project/Task/Work Unit No. 11. Contract/Grant No.
12. Sponsoring Organization Name and Address ADPE Selection Office Department of the Navy Washington, D. C. 20376			13. Type of Report & Period Covered 14.

15. Supplementary Notes

16. Abstracts

This technical paper discusses the Debugging features of COBOL 74 (American National Standard Programming Language COBOL X3.23-1974). The formal COBOL specificationsup until COBOL 74 totally ignored the problem of debugging programs at the source level. This paper discusses some of the methods by which the various vendors who produce COBOL compilers attempted to break the problem by providing language extensions to their compilers. It goes into the COBOL 74 debugging features in great detail as well as provides numerous examples of using debug lines in the source code debug procedures in the Declarative portion of the source program.

17. Key Words and Document Analysis. 17a. Descriptors

COBOL
Validation
Software
Audit Routines
Verifying
Compilers
Standards
Programming Languages
Portability

17b. Identifiers/Open-Ended Terms

17c. COSATI Field/Group 09/02

18. Availability Statement	19. Security Class (This Report)	21. No. of Pages
Release unlimited.	UNCLASSIFIED	6
	20. Security Class (This Page) UNCLASSIFIED	22. Price